Clean R code cheatsheet

Clean code in R workshop, eRum 2018

Ildi Czeller and Jeno Pal

May 2018

*"One difference between a smart programmer and a professional programmer is that the professional understands that **clarity is king**. Professionals use their powers for good and write code that others can understand."*

1. Use meaningful names

Use intent-revealing names

```r
nd <- 3
```

```r
number_of_days <- 3
```

Use names that you can pronounce

```r
rows_w_miss_val <- df[!complete.cases(df), ]
```

```r
rows_with_missing_values <- df[!complete.cases(df), ]
```

Use verbs to name functions

Functions *do* something with inputs. As such, choose a name that reflects what it *does*.

```r
client_data <- function(...)
```

```r
get_client_data <- function(...)
```

Use names that are easy to distinguish

Avoid confusion stemming from using both single and plural of the same name

```r
rows_with_missing_values <- df[!complete.cases(df), ]
```

Don't use number suffixes to distinguish variables

Use one word for one concept

`get`, `retrieve`, `fetch` are synonyms. Pick one if two functions perform the same action.

```r
get_client_data <- function(...)

fetch_location_data <- function(...)
```

```r
get_client_data <- function(...)

get_location_data <- function(...)
```

Do not overwrite variables

```r
customers <- delete_rows_with_missing_values(customers)
```

```r
complete_customers <- delete_rows_with_missing_values(customers)
```

Choose names that do not conflict with base functions or keywords

Do not use noise words

? `dt_customers` vs `customers`

Avoid magic numbers

Use named variables instead

2. Functions

Don't repeat yourself (DRY)

A function should do precisely one thing

Extract code to function to name it

Even if you don't plan to reuse the code in more than one places.

Avoid too many parameters (> 3)

Pass all parameters to a function as arguments

This makes functions self-contained. Very rare exception: global constants with a naming convention that is easy to follow (e.g. `ALL_CAPITAL_LETTERS`).

Clearly separate functions with side effects and functions with a return value

Save figure vs return figure: create separate functions.

Organize your functions from top to down in abstraction levels

Main functions should come first, lower level functions that they use come below them.

3. Comments

Explain with the code itself rather than with comments

```r
# calculate customer lifetime value
c_ltv <- calc_cust_LTV(cust_data)
```

```r
customer_lifetime_value <- calculate_lifetime_value(customer)
```

4. General refactoring tips

First write a working code, then make it cleaner (= refactor)

Boy scout rule: if you modify something, think about leaving it a bit cleaner than it was

If you have to touch a piece of code for any reason, consider refactoring it as well. You may have better sense/ideas of your code later than writing it first even if you struggled to write clean code in the first place.

Refactoring is not writing it from scratch again

Small refactors are more effective than complete rewrites. Guarantees that your code works the same as before and that you can move on quickly.

Code should be readable also to people not familiar with R

You don't have to overdo refactoring (dont' create a new function for `dplyr::filter`), however, hide somewhat cryptic parts to named piece of code (example: `apply(dt, 2, fun)`)
Read code as a book on a high level.

Scripts should be self-contained

do not require the manual sourcing of another script or libraries

Remove dead functions, do not leave commendted-out code

Literature

Robert C. Martin: Clean code.